Bachelor Informatica

Universiteit van Amsterdam

Task Migration in S-Net

Stefan Kok, MSc, 6084044

June 12, 2012

Supervisor(s): Dr. Clemens Grelck (UvA), Raphael 'kena' Poss, MSc (UvA), Merijn Verstraaten, MSc (UvA)

Signed: Dr. Dick G. van Albada (UvA)

Abstract

S-Net is a coordination language designed to simplify parallel programming. It is a language based on streaming networks that has a runtime system running different tasks that either do computations or are directing streams to create parallel structures in such a way that data can flow through the network with minimal delay. The goal of this thesis is to create a framework for the runtime system that is able to migrate tasks. Task and process migration has been a research topic since the eighties, but the difference between this research and the research done in this thesis, is that the runtime system has more information about the program that is executed in the runtime system. The S-Net runtime system knows the structure of the S-Net program and this can be used to decide whether tasks should be migrated or not. This research will look into different strategies to migrate tasks. These strategies will also use some information about the structure of the S-Net program. However, the results show that the new implementation does not perform better than the previous implementation of the S-Net runtime system. For now, the framework created in this research does not rise to the expectations.

Contents

CHAPTER 1

Introduction

Software must now rely on on-chip parallelism instead of frequency increases to achieve increased performance [1, 2, 3]. Following Moore's Law, transistor sizes are still shrinking but clock frequencies are not increasing and thus most manufacturers have chosen to put multiple cores on a single socket in an attempt to increase computing power. There is no linear relation between increasing the number of cores and performance, because creating programs that run efficiently on multiple cores is not a straightforward task. One reason is the limitations of other components besides the cores. For example the latency to access memory has increased relative to the core pipeline cycle time, which is called the memory wall[4].

Another reason is that most programs are sequential programs and do not use the extra cores that are available. Until some years ago, parallel programming has been done only in very specific areas. If performance of programs is to be increased, a program should make as much use of the parallel processing power as is available. This will require new programming tools that are better able to utilise multiple cores.

S-Net [5] is a declarative coordination language that is designed to simplify parallel programming. S-Net itself is not designed to do computations, it uses sequential blocks written by an application programmer in some language like C. This results in a separation of programming the application and parallelising program. Thus the concurrency programmer does not need to have specialist knowledge about the algorithms used for the application itself. This task can be done by the application programmer.

S-Net consists of box languages, compiler and runtime. The compiler generates a program that creates tasks. The run-time system is responsible for the dynamic management of tasks, including handling communication events over streams and the distribution of tasks over execution resources.

S-Net is still in development and there are a number of parts in the S-Net runtime system that could possibly be optimized. One such part is the division of tasks among different cores. The prior implementation is straight-forward, a task is placed on a core according to a round-robin algorithm. Furthermore, once a task is placed on a worker, it will not be migrated. This could result in a sub-optimal usage of CPU-time when multiple high-demanding tasks are spawned on the same worker, while other workers are idle waiting for work.

To ease the issues of load imbalance, further research must be done to equip S-Net with "intelligent" placement strategies. However, there was no support for task migration in S-Net prior to our work, but may be required. In this BSc thesis we set out two goals. The first is to introduce infrastructure in S-Net that eases task migration, which would help in further research in this area. As later described, this foundational goal will also involve rewording some basic S-Net abstractions in collaboration with the rest of the S-Net ecosystem. As a second goal, we intend to illustrate, using example placement oracles, that our proposed infrastructure makes it easy to achieve better performance than the current static placement (Poss, Raphael. MSc.) . In chapter 2 a description is given of the S-Net programming language. Furthermore, the runtime system previous of this research is discribed in this chapter. Chapter 3 gives a short overview on task migration. Next chapter 4 gives a description of what the goals are for this research.

Then in chapter 5 the implementation of the goals is explained. After this, chapter 6 describes the experiments done with the S-Net runtime implementation implemented in this research. Furthermore the results are shown in this chapter. The results will show that the framework created during this research for now is not beneficial to the performance of the runtime system. The last chapter, chapter 7 discusses the results and what can be concluded.

CHAPTER 2

Background

2.1 S-Net

S-Net [5, 16, 6, 7, 8] is a declarative coordination language. It is designed to ease the transformation of existing sequential application code into a concurrent application that can exploit parallelism in a computing platform. S-Net is not intended to do computation, but describes the data dependencies. The actual computation is done by the box code.

S-Net is inspired by the theory of streaming networks. It consists of a network of boxes. These boxes have a single typed input stream and a single typed output stream (SISO). Within a box the application code is executed. This is sequential code written in a arbitrary language like C or SAC [9]. Networks can be created by using network combinators that can be applied on both boxes and networks. These will create new SISO components.

A data-item in S-Net is called a *field*. Records group fields together. Data is then communicated across streams through the use of these records. Computations are done on single records. After computation either zero records or one or more records can be passed to the output stream of the box.

As mentioned before, the application is separated into a coordination part done by S-Net and a computation part. This creates the possibility to breakup or separate these two tasks and let an application programmer whose talents lie with programming algorithms, build the application itself and let a concurrency engineer design the S-Net code that creates an application that runs the program concurrently on different nodes.

In the rest of this section first the type system of S-Net will be described in subsection 2.1.1. This includes a description of subtyping, type signatures and inheritance. Next, in subsection 2.1.2, boxes, filters and synchrocells are described. Furthermore, in subsection 2.1.3 the different network combinators will be explained.

2.1.1 Type System

The type system in S-Net uses subtyping on records. A type in S-Net is a non-empty set of record variants. Each record variant is a set of record entries, which can be empty.

Record entries can be one of three different types. Each type is a label-value pair. There are fields, tags and binding tags. Fields are labels that include data which can not be accessed by S-Net. The data can only be manipulated within a box.

Tags and binding tags also have labels and an integer value connected to it. Tags and binding tags are publicly accessible for the S-Net system.

Types consist of record variant the are separated by a `"|"`. Record variants are enclosed by `"{}"`. These record variants can contain fields, tags and binding tags. Fields are just strings, while tags are strings enclosed in `"<>"` and binding tags use a `#` in front of the string to separate them from normal tags.

An example of a type is the following:

```
{<#circle>, <x>, <y>, radius} | {<#square>, <x>, <y>, width}
```

The example consists of one type containing two record variants, a circle and a square, which are denoted as binding tags. Both record variants have an x and a y tag and the circle has a *radius* field, while the square has a *width* field.

Record Subtyping

In S-Net a subtype is a type s that is a more specific type than its supertype t. There is a formal specification of a subtype. There are two definitions for subtyping, one for record variants and one for types. Let $BT(x)$ be the set of binding tags within variant x then,

1. A record variant v_1 is a subtype of a record variant v_2, $v_1 \sqsubseteq v_2$, *iff*: $v_1 \supseteq v_2 \wedge BT(v_1) = BT(v_2)$
2. A type t_1 is a subtype of type t_2, $t_1 \sqsubseteq t_2$, *iff* $(\forall v_1 \in t_1 \exists v_2 \in t_2) v_1 \sqsubseteq v_2$

What this means is that a subtype v_1 of a record variant v_2 has the same binding tags and v_1 has the same set of record entries as v_2 plus additional record entries. While a subtype t_1 of a type t_2 has a subset of record variants that are also in t_2. Thus a record variant with less record entries is likelier to be a supertype than a record variant with more record entries. For types the opposite holds, a type with more record variants is likelier to be a supertype as a type with less record variants.

Type Signatures

A type signature is a mapping from a non-empty set of types to a non-empty set of types. It describes the types that can go into and out of a box or network. An example is:

```
{a, b} | {c, d} -> {<x>} | {<y>}, {e} -> {z}
```

Here there are two mappings separated by a `|`. The first mapping can accept two different types of record variants, `{a, b}` and `{c, d}` as input type and has either `<x>` or `<y>` as output type. While the second mapping has `{e}` as input type and `{z}` as output type. The first record variant can also be rewritten as:

```
{a, b} -> {<x>} | {<y>}
{c, d} -> {<x>} | {<y>}
```

A signature with only one input variant will be called a normalized type signature. The output type cannot be normalized in the same way as the input type.

Flow Inheritance

Using subtyping brings a problem to what should be done with fields and tags in the subtype that does not belong to the input type signature. For example, with the type signature:

```
{a} -> {x}
```

and an incoming record:

```
{a, b}
```

What should be done with field `b`? It is possible to just discard this field, but that would lose the advantage of re-usability with subtyping. In S-Net the choice is made to append a copy of any fields or tags from an input record, that does not appear in the input type signature, to the output record. In the example given, this would result in the output record looking like the following:

```
{x, b}
```

But this will cause another problem, namely what to do when the field or tag is in the output type signature? For example, `{a} -> {x, b}` would result in a duplicate of `b` in the output record which is not possible. The solution is to keep the field or tag from the output record and discard the version of the input record.

2.1.2 Entities

Box

A box has a single input and a single output stream and is stateless. Each box has a unique name. A box runs an external code possibly written in another programming language, i.e. it runs the code written by the application programmer. They have a box signature which is a normalized type signature. The order in which all input arguments are given, is defined by the order of the type signature. This is because for example box code implemented in C for example, the order of input arguments is important and thus the problem is solved in this matter.

Filter box

As with normal boxes, filter boxes are also stateless. There are a number of roles that the filter box is used for, such as:

- removing fields or tags from records
- duplication of fields or tags
- adding some tag
- splitting records
- some simple computations on tag values.

A filter box has exactly one input variant, zero or more guarded actions and one unguarded action. A guarded action consists of a boolean expression, if the expression is true, the guarded action will be performed, otherwise the next action will be performed. The unguarded action is the last action in the sequence and if all boolean expressions are false, the unguarded expression will be executed. Some examples of filters are:

1. `[ {a, b} -> {a} ]`
2. `[ {a, b} -> {a}, {b} ]`
3. `[ <a> -> <a>, <b=a> ]`

The first just removes `b` from the record. The second splits the record into two records and the last splits the record into `<a>` and copies `<a>` to `<b>`. For a more elaborate description of box functions we refer to [8].

Synchrocell

Synchrocells are the only entities that have a state. They are used to combine multiple records into one record, as this is an aspect of the filter box which it doesn't do. The reason a synchrocell has a state is that it has to wait for multiple records to pass the synchrocell and wait for the correct types of records to come in and combine them into one new record. An example of a synchrocell is the following:

```
[| {a, b}, {c, d} |]
```

This synchrocell waits for two records, one that matches `{a, b}` and the other `{c, d}`, it can also match subtypes of these record variants. When one of these record variants is matched, it is stored and waits for the other record that matches the other record variant in the pattern. All other incoming records will be passed directly to the output stream, including records that are already matched by one of the record variants. Once all records in the pattern are matched, they are combined into one record and the synchrocell will stop working, i.e. it will pass any incoming record directly to the output stream.

2.1.3 Network Combinators

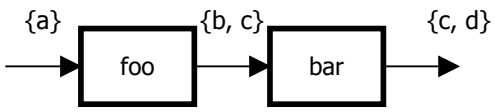
Network combinators are binary operators that can be applied to boxes. The result is a network with one input and one output stream. A network can also be used as an argument for the combinator. From now on, the word *components* [7] is used instead of boxes and networks on which combinators are applied to.

There are 5 combinators. 3 of these have a both deterministic and non-deterministic version of the operator. These are the star, feedback and parallel combinator. A deterministic operator preserves the order of the records, this will result in records that are the first to come in, are the first to go out. Deterministic combinators are defined in the S-Net language by using two combinator symbols instead of one, ie. `**`, `||`, `!!` instead of `*`, `|`, `!`. The serial and feedback combinators only have a deterministic version.

Serial Combinator

The serial combinator `..` puts two components into a sequence. The output stream of the first component will be the input stream of the second.

```
net example {
    box foo ((a) -> (b, c));
    box bar ((b) -> (d));
} connect foo..bar;
```

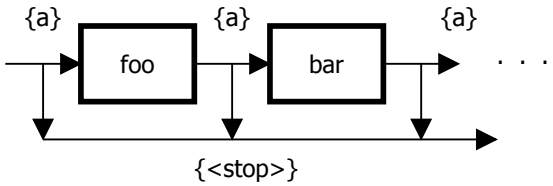


In the example, there are two boxes, `foo` and `bar`. The type signature for this network will now be `{a} -> {c, d}`. Note that `c` is part of the output signature of `foo` but not the input signature of `bar`. It is not used in `bar` and as described under subsection 2.1.1, it will be appended to the output of the `bar` box.

Star Combinator

The star combinator `*` is the dynamic counterpart of the serial combinator using only a single component. A star combinator has a component and a stop condition as arguments. As long as the output of the given component is not equal to the stop condition, the output record will go through a copy of the same component, thus creating a sequence of the same component. A note to make is that a record could immediately be passed to the output stream if it meets the given stop condition.

```
net example {
    box foo ((a) -> (a) | (<stop>));
} connect foo*{<stop>};
```

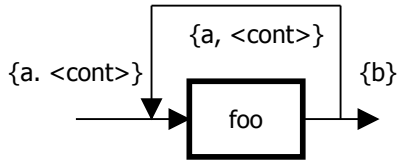


In this example, any incoming record with type signature `{a}` will go to the input stream of `foo`. Afterwards, the record is checked, if the type signature is `{a}`, it will be forwarded to a next instance of `foo` and if it is `{<stop>}` it will go to the output stream.

Feedback Combinator

The feedback combinator \ has some similarities with the star combinator. A record will go through a given component and stops when the condition is not met. Here there are three important differences in comparison with the star combinator. First the record will pass the component at least once. Furthermore the condition given determines continuation instead of ceasing. Finally, there is only one instance of the component created, ie. if the record does met de continuation condition, it will go to the same instance of the given component instead of a new one. The feedback combinator can be seen as a sort of loop.

```
net example {
    box foo ((a, <cont>) -> (a, <cont>) | (b));
} connect foo\{<cont>};
```

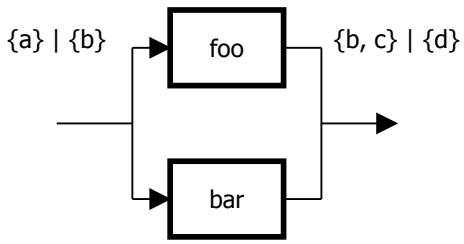


This example shows a network for a feedback combinator. Box `foo` will either output `{a, <cont>}` or `{b}`. The condition to continue is `<cont>` and if the output of `foo` is `{a, <cont>}`, the record will loop back to the input stream of `foo`, otherwise it will write `{b}` on the output stream.

Parallel Combinator

Just as the serial combinator the parallel combinator | is a static combinator. It has two components as arguments and puts these two components in parallel with each other. An incoming record will then go either to the first or second components, based on the type signature of the record and component. If the records type signature matched both components type signatures, it will go to the one that matches the records signature the closest.

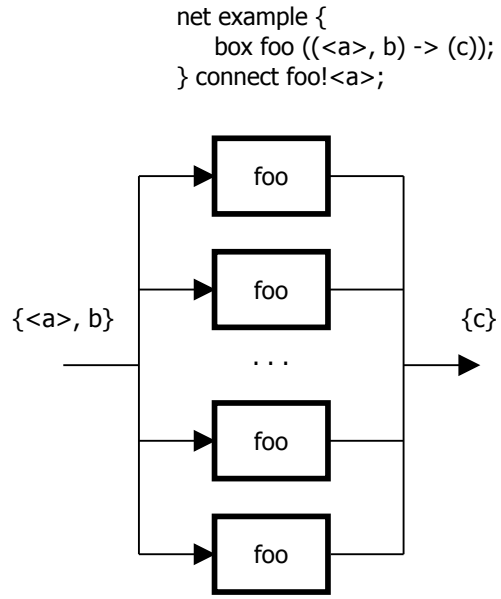
```
net example {
    box foo ((a) -> (b, c));
    box bar ((b) -> (d));
} connect foo | bar;
```



The parallel combinator in this example puts two boxes `foo` and `bar` in parallel. The input type signature will then be `{a} | {b}`. All records with `{a}` will be forwarded to `foo` and records with `{b}` to `bar`. The output will either be `{b, c}` or `{d}`, depending on which box did the calculations.

Split Combinator

The split combinator ! is the dynamic counterpart of the parallel combinator and just as the star combinator is only applied to one component. The second argument is an index tag. For each value of the index tag, it creates an instance of the component. Incoming records are then routed to the instance that has the same tag value.



The example for the split combinator uses the index given in `<a>` and forwards the record to the copy of `foo` with the same index.

2.2 S-Net Implementation

2.2.1 Runtime system

In this subsection a global explanation will be given on how a task runs in S-Net. In figure 2.1 a graphical example is given on how such a task runs.

First a scheduler chooses a task to run. It will then go to the task wrapper that runs the task in a loop. The task will first do a read to fetch a record from the input stream. If no records are on the input stream, the task is suspended and the scheduler will run another task while the task waits for an incoming record.

When there is a record on the input stream, the task will be set to ready and the scheduler will run the task when other tasks with higher priorities have finished or are suspended. The read will then return a record to the task wrapper, which calls the box function. The box function will do some calculations on the incoming record and writes its output to the output stream. This in turn can also be suspended, for example when the output stream buffer is full. It will be suspended just as with the read function and at some moment in time the scheduler will again run the task if the output record can be written to the stream. Finally the box task returns and the task wrapper will begin another iteration.

2.2.2 Threading layer

LPEL stands for Light-Weight Parallel Execution Layer [10]. LPEL replaces the original threading back-end for the S-Net run-time system [11]. In the standard implementation each task runs in its own PTthread. Instead, LPEL uses workers that each run on a single core, which run using a PThread. Furthermore, it uses user-threads called tasks that run on the worker. The advantage of this implementation is that there is a potential to control where tasks should run and as a result, the run-time system such as S-Net, could do a better job because it has more information about the program. The reason for this is that PThreads are kernel-threads and where and when threads run is decided by the the operating system. However, the potential of scheduling tasks by LPEL itself is not implemented here and is described in the following chapters as that is the research topic of this thesis.

This implementation can be very inefficient, the scheduling algorithm of the operating system may work well for certain applications, but it is not optimized for that specific application.

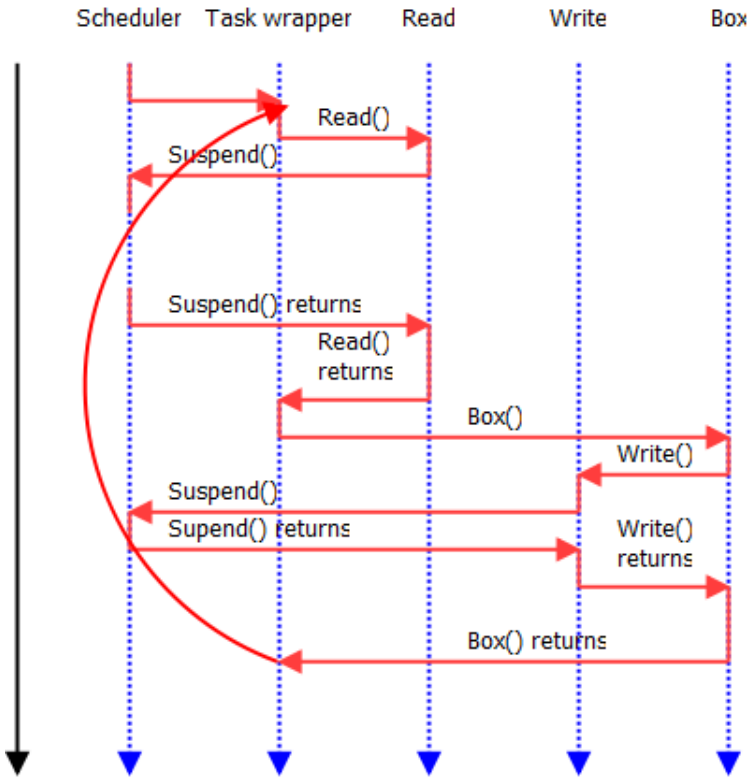


Figure 2.1: This is an example of how a task runs in S-Net. First the scheduler will choose a task to run. Then it will read from the input stream to fetch an incoming record. If the stream is empty it will wait and the scheduler will run another task, while the suspended task waits on an incoming record. When the task has a record on the input stream, the scheduler will run it once other tasks that have priority have finished. It will then run the box function to do the calculations. After the calculations are done, it will write to the output stream, which may be suspended as well. Finally when the box function returns, the next iteration of the task will be done.

Furthermore, each task runs in its own thread, which means that if there are a lot of tasks running, the OS has to do a lot of context-switching, which is expensive.

A solution is to have a combination of kernel-threads and user-threads. Each core runs its own kernel-thread, a PThread. The PThread runs a worker that runs user-threads called tasks. A worker consists of a scheduler, a priority queue and a mailbox. Communication between tasks and a worker is done using the mailbox. For example when a new task is created it sends a ready message to that mailbox.

The scheduler uses a the priority queue to decide which task should run. Tasks that are not in the queue are tasks that are waiting for IO. A task could, for example, be waiting for data to come on the input stream.

Streams in LPEL are connected to two tasks. A task that produces data and a task that consumes data. When a stream is used it is connected to two stream descriptors. These stream descriptors are wrappers to bind a stream to a task. If a read is done and there is no data on the stream, the task connected to the stream by the stream descriptor will be suspended. The stream descriptor is also used here for the wake-up. The task can be woken if there is data on the stream, by getting the consuming stream descriptor and setting the task to the ready state and send a message to the worker.

CHAPTER 3

Related Work

3.1 Static Load-Balancing

Static load-balancing [12] aims to divide the work over multiple processors before a program starts computation. This might not be the most effective strategy as workloads may fluctuate during execution. Also, the state of the system may change, this might cause the program to get bogged down because other programs are running as well, or communication between different processors slows down. Some strategies for initial placement [12] are:

Round robin, here each task is assigned to a next processor, after all processors are assigned to a task, the next task is assigned to the first processor and so on,

Random, tasks are randomly assigned to different processor,

Recursive bisection, aims to divide the problem into subproblems and dividing an equal amount of computation time over the different processors, with minimum message-passing.

3.2 Dynamic Load-Balancing

Dynamic load-balancing, [12] unlike static load-balancing, aims to divide computational time over different processors during program execution. This means that tasks can be migrated to other processors if a processor has too much work, while others are idle or have very little work to do. Load balancing can be done centralized or distributed. A centralized system uses a master slave system where other 'worker' processes request tasks from a master. The master has a task queue and decides what tasks are given to which worker.

A distributed system has multiple 'task pools' where slaves can ask for work. A fully distributed system does not have any such pools, just workers that communicate with each other if they need work or need work to be taken over by others. A distributed load-balancing system has a number of policies to decide migration [12], namely: a transfer policy, a selection policy, a location policy and an information policy. A transfer policy decides if a node is suitable for migration. This holds for both sending and receiving nodes. An example of such a policy is a threshold policy with high-loaded, middle-loaded and low-loaded nodes [13]. Where low nodes are nodes with little work and are target for receiving work. Medium nodes have enough work and are not used for placement. High nodes have too much work and are targets for sending worker.

The selection policy decides which tasks are to be migrated. There are some different strategies for selection, for example random selection or choosing tasks which get little running time.

The location policy is used for finding a good "transfer" partner. In a master-slave system this is done by finding the information at a master, in fully distributed systems this is done by communicating with each other. This communication is expensive and often is limited to, for example, only neighbor-nodes [12].

The information policy is responsible for collecting information about the loads of different workers. As already noted, communication is expensive and this policy decides when the worker sends information to other workers and to whom. This can be done using different methods, for example only communication when a worker is in a state that it needs to remove tasks or fetch new tasks. Another method is to periodically send information to other workers.

3.3 Work Stealing

Work stealing is done on shared-memory multiprocessor systems [12]. As the name implies, workers try to steal work from other workers. Stealing workers (thieves) do not have enough work and try to find heavy loaded workers (victims) that do. One aspect to keep in mind is that locality can play a role in shared-memory systems. This makes it more interesting for a stealing worker to find workers that are close to the thief. This could potentially decrease communication time as the memory can be accessed faster compared to tasks stolen from processors that physically lie further away.

Also the strategy of which task to steal is important [12]. Stealing a task might cause problems because it is already in the cache of the core. Migrating it would result in cache misses for the core as it needs to load other data to it. Mostly tasks are stolen from the back of the queue as these tasks will take a while to be loaded for running and will not cause any problems when migrating them.

CHAPTER 4

Theoretical Objectives

The goal of S-Net is to better use the possibilities of parallelism on multi-core machines and multi-node systems. S-Net programs are dynamic in the sense that if needed, networks can grow and shrink. This means that placement can not be decided before the program starts running. Another important point is that the amount of computation time for a box task is often dependent on the input record. S-Net should thus have a runtime-system that actively tries to balance the workload over the different workers.

As discussed in chapter 3, a solution for balancing work is to use an algorithm like work-stealing. However the S-Net runtime system has some knowledge of the program. First of all there is knowledge about the different tasks. A distinction can be made between control tasks and box tasks. Here control and box tasks are all the entities named in subsections 2.1.2 and 2.1.3, furthermore this also includes a collector/merger used to gather data from different streams such as in the parallel combinator or split. Next there is also knowledge about records, fields are opaque to S-Net, but tags and binding tags are not. This might help decide how long a box task runs. Using this knowledge could have advantages over the scheduling techniques used by an operating system, this is another reason why the LPEL implementation is used instead of PThreads.

Locality is another advange in the S-Net system. A program is a network of boxes, this network has a certain topology which can be used to determine where to place tasks. This could be important because multi-core systems might have shared memory, but cores that are further may apart have a higher communication time. Instead of migrating tasks, migrating parts of the S-Net network, might decrease communication time. For migration this means that complete parts of the network are migrated together to the same worker.

4.1 Migration framework

This section will discuss the framework for migrating tasks. This section will explain what should be done to create a framework for task-migration.

Figure 4.1 shows a diagram of how task migration can be achieved. Instead of the scheduler spawning a task, it first spawns an initialisation function. This is only done on creation of a task.

After this, the main function is called. Reads and writes are done in the task and when an iteration [1] is finished, instead of looping to the top of the task, the task respawns itself. Instead of an iterative function, this looks more like a recursive function. The result is that at the end of one iteration, the complete state is known and thus can be given as an argument when spawning the new task.

[1] An iteration is one run in a loop

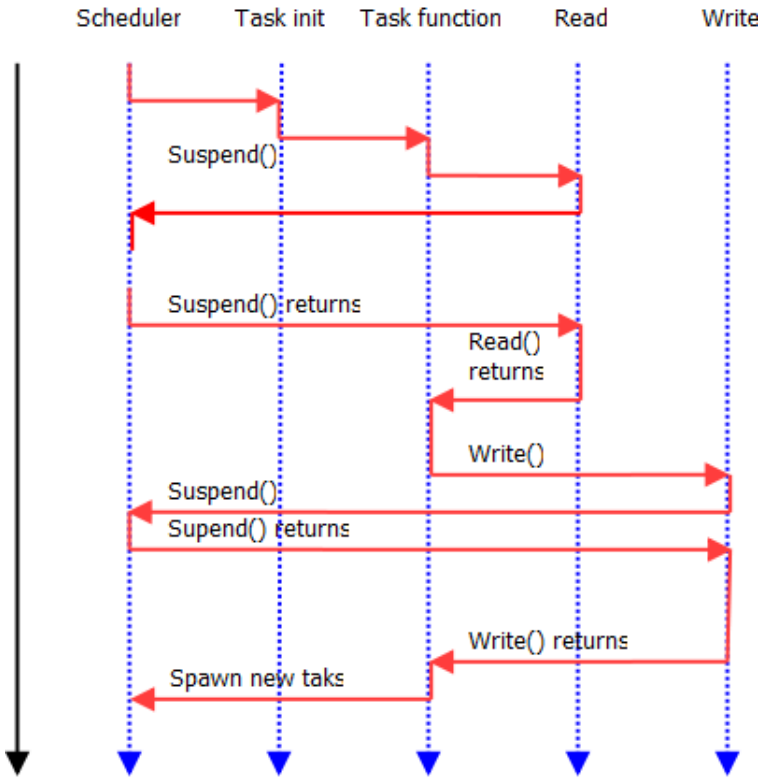


Figure 4.1: The framework that uses task-migration starts with the initialization of a task. After this, the main function is called, which does a read. It then suspends if there is no record on the input stream and other tasks can then run. When the tasks is ready and the scheduler assigns this task to run, it then does some computations and writes to the output stream, which may also suspend. Finally the task calls a spawn function, which spawns a new task, running the same main function.

4.2 Placement Scheduler

A placement scheduler is used to decide on which workers a task should run. This section will discuss which implementation we chose and why. One consideration to make for a placement scheduler whether it should be a synchronous or an asynchronous system. A synchronous system would be an implementation where during the respawn the task asks the scheduler whether it should migrate or not. An asynchronous implementation has an active scheduler. The scheduler is running in parallel in with the tasks and sets the parameter on which worker the task should respawn asynchronous of the respawning.

This might result in a scheduler deciding a task should migrate, but as the task is still running it does not do anything with this information. Then the scheduler decides the task should be kept on the same worker and the task finally respawns on the same worker. Figure 4.2 shows a graphical representation of this implementation.

An advantage of this implementation is that the scheduler can make decisions based on the whole state of the runtime system, instead of only the state of one task. Thus, it can be decided to migrate multiple tasks at once.

4.2.1 Algorithms

As already discussed in this chapter, the S-Net scheduler has some advantages over more naive schedulers used by an operating system. Distinguishing between box tasks and control tasks is one such advantage that is implemented. Control tasks are tasks that are used to direct the

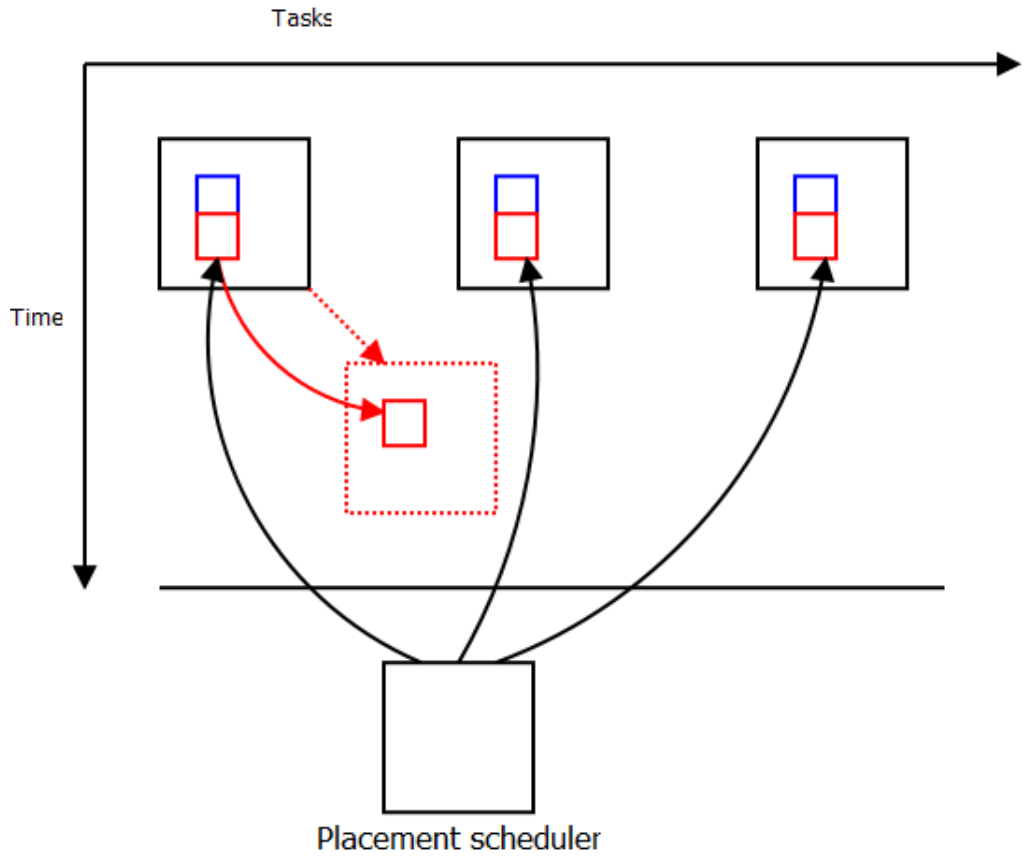


Figure 4.2: An example of how the placement scheduler works. Each box is a task running, containing a set of variables. This set of variables also contains the current and new location of the task. The scheduler decides where the task should do its work and will write this to the new location variable. Once a task is finished with an iteration and the scheduler has decided the task should migrate, a new task is created and placed on a different worker.

stream flow, heavy computations are done in boxes. Creating workers that only handle short control tasks or heavy box tasks might increase the flow of the network. Control tasks are able to continuously pass records and create new entities, without waiting on computational intensive box tasks.

Another method to use task type information is to use a priority queue for scheduling. For example, we could give control tasks a higher priority as they are short tasks and take up little computation time.

Initial placement is another aspect of placement. This part is done without the use of the placement scheduler, as the placement scheduler only does scheduling using active tasks, that are already created. The choice is made to use some form of locality for initial placement. Some combinators initially have only a dispatcher running, namely the split and star combinators. When created these are very small tasks, but can create a large amount of new tasks. On the other hand, the parallel combinator is created statically and this can be used to decide where tasks after the parallel combinator should be created. The parallel dispatcher is created on the same worker as the combinator or box that writes to the parallel dispatch input stream, but the second component of the parallel combinator is created on a different worker. Furthermore for the split combinator any newly created instances of the network within the split are placed on a new node.

The placement scheduler itself is implemented as two different algorithms. The first makes a random choice to migrate and where to migrate. The second uses the waiting time of a task. A task has a state, created, running, ready, blocked, mutex and zombie. A task should either be

waiting on a read or write, or it should be running. A task which is ready to run but is waiting for other tasks to complete could possibly run on another worker.

Each task has a ready time: $ready_t$. This is the amount of time the task was ready for a certain period. This period is based on the amount of switches between the task switching from the ready state to another state. The average of this ready time is:

$$\mu_{ready} = ready_t/n$$

This number is then used to calculate the average ready time for each worker. This is done by taking all m tasks that are in a ready queue as follows:

$$\mu_{worker} = \frac{\sum_{i=1}^{m} \mu_{ready}}{m}$$

A threshold function τ then decides if tasks should be migrated. This happens when $\mu_{ready} \geq \mu_{worker} \cdot \tau$. This task is then put in an ordered list. If the task is placed at the back of the list (is closest to the threshold), but the list is not big enough, it will not be migrated. This ensures that the number of tasks that is migrated is limited, to prevent the scheduler from excessive task migration.

Workers that are waiting, i.e. do not have a running task, are candidates for the tasks that are selected for migration. These workers do not take part in the procedure of selecting tasks to migrate[2].

[2]A worker that is waiting, should not have any tasks in its queue as these are tasks that are ready to run

CHAPTER 5

Implementation

5.1 Migration framework implementation

In this section a description is given on the changes made to the original implementation of S-Net and LPEL. The changes to make a framework for task migration. As mentioned in section 4.1 tasks should re-spawn instead of iterate.

To create a framework that removes the iteration within the tasks as described in chapter 4, the different entities are changed. The following entities are changed:

- The box task
- The collector
- The feedback combinator
- The filter
- The parallel combinator
- The split combinator
- The star combinator
- The synchrocell

Each of these entities have a main task[1] and an initialization task. The initialization task is responsible for initializing some parameters like stream descriptors for the input and output streams. The main task is responsible for one iteration of the task. The iteration will either terminate or a re-spawn function is called.

An entity runs in an LPEL task which calls a function to run the code in S-Net. This consists of a loop that keeps running until a flag in the arguments is set to stop. This function calls the actual entity:

```
RunTask(entity ent) {
  do {
     ent.run = false;
     ent.function;
  } while(ent.run);
  delete ent;
}
```

[1]The feedback combinator has multiple tasks. The description of the feedback combinator can be found in appendix B

There are two more spawning functions `SNetThreadingInitSpawn` and `SNetThreadingReSpawn` added. The first spawns the initial task. The second is called by the entity when a task should continue to run. It compares two variables, the first is the id of the worker on which the task runs and the second is the worker the task should migrate to, which is set by the placement scheduler. If these two ids are different, the task should migrate and a new task is created and the current task stops. If the worker ids are the same, the flag in the entity arguments is set to run and a next iteration is done.

This implementation does not have a full respawn framework, as a task is not respawned if it keeps running on the same worker. This is an easy and effective implementation which simulates the circumstances described in chapter 4.

However this new implementation introduces a problem with stream descriptors. A stream descriptor is bound to a stream and task. When a task migrates, a new task is created. The result is that stream descriptors have become useless, because they are not bound to the correct task anymore. There are some possibilities to solve the problem, one is to either update the stream descriptors, another is to destroy all old stream descriptors and create new ones. These solutions are possibly expensive as they require iterating through different stream descriptors[2]. The solution chosen here is to update the stream descriptor when it is needed. The functions to read and write to streams use the stream descriptor to get the task that is connected to the stream. In these functions it is possible to find the current task that is running, which is the task that calls the read or write function. This can then be used to update the stream descriptor.

5.2 Placement Scheduler

The placement scheduler is implemented as a task on a worker. This worker is only used for running the placement scheduler and runs in the background, like the input and output manager of the runtime system. Furthermore, the scheduler consists of a loop which calls the placement algorithm and then yields itself. This creates the possibility for the worker to fetch messages from the mailbox, most importantly a message that tells the worker to terminate.

There are a number of possibilities on how tasks are initially scheduled. As already discussed in section 4.2 a task can be scheduled to run using a priority queue, but there is also the possibility to use segmentation between control and box tasks. They both use a flag to decide either which priority the task has or on which worker the task should be placed.

As the priority queue already exists in the LPEL code, this part does not have many changes to the LPEL code. The only difference is that the function to create a task has an extra parameter for the priority.

The system for segmentation between control and box tasks is implemented by using a list for workers that run box tasks and a list for workers that run control tasks. A parameter determines the number of workers running control tasks.

5.2.1 Implementation of the algorithms

We implemented two different placement scheduler algorithms, random scheduling and ready-time scheduling. The random scheduling algorithm randomly assigns tasks to new workers. It goes through each worker using an iterator that goes through all tasks that are in the ready state that are in the priority queue of the task-scheduler of the worker. It first decides using a threshold if a task should be migrated, after this the new worker is decided at random.

The ready-time scheduling algorithm is based on ready time. Each task has a set of parameters that store the relevant statistics. There is a variable that keeps track of the time that a task went from some state to ready. After the task switches back to another state, the time is taken again and the difference is calculated, this is the time that the task has been ready.

There are a number of choices for keeping track of the ready time. One is to just add ready time and take the average each time it is needed. This would create a problem that the state of the worker changes over time and for example long waiting times that are long solved still have some weight in the average. The other solution is to just take one measurement. The problem

[2]The problem is mostly the collector which has a dynamic set of stream descriptors

with this solution is that it is a small snapshot and it could be that the time the task is in ready mode is small all the time except for one moment, which causes the task to be migrated.

The solution used is something in between the previous two. The average is taken over a maximum number of times that the state of the task switches from ready to another state. A sliding window is used to deal with the problem that only one measurement is taken. This is done using a binary sliding window in the task. This stores the sum of the measured ready times. Every n times, the other value in the sliding window is used. Every $n/2$ times the other value starts taking measurements. A graphical representation is shown in figure 5.1.

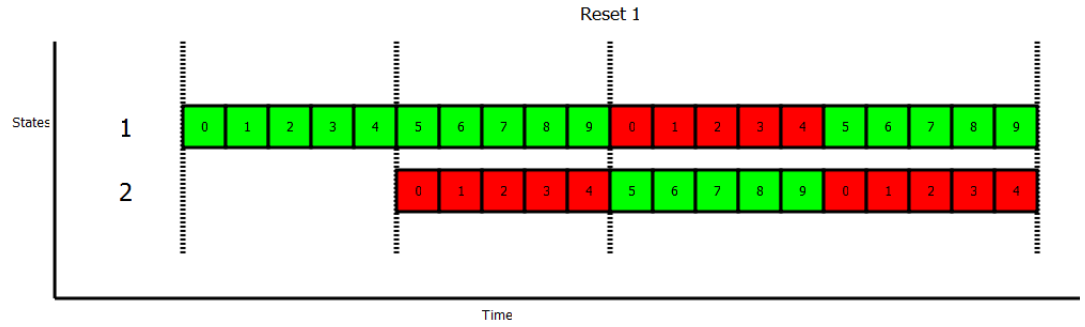


Figure 5.1: The sliding window algorithm first adds time to the upper measurement. Green/light implies that it is written to and also used by the placement scheduler for the average ready-time. Red/dark means that it is written to, but is not used by the placement scheduler. At step 5, the lower array is also written to. When the first measurement has summed ten different measurements that a task has been ready it is reset and the second measurement is used by the ready-time placement algorithm.

The other part to keep track of is if the worker has been idle, waiting for tasks to finish IO. A flag has been added to the workercontext. This is set to one when a worker has been waiting, the placement scheduler sets this flag to 0 on the first encounter of the worker.

The first step the placement scheduler takes, is to find which workers are receivers and which workers are senders. This is done in two ways, a worker is a task receiver if the flag is up and the other possibility is that there are no tasks in the queue. All other workers are potential task senders. For each of these sending workers, the average ready time is taken, by summing the ready times of all tasks in the priority queue and dividing it by the number of tasks in the queue.

After this tasks are selected for migration. There are two methods to decide which tasks should be migrated. First every task in the priority queue is compared with the average ready time of the worker its running on. If the average ready time of the task is higher than the time of the worker multiplied by some value, it is selected for migration. Next there is a fixed sized ordered array. The first task has the highest average ready time. If a task does not fit in the array, it is not selected for migration. This also reduces the number of tasks that can be selected for migration.

In the next step, for each receiving worker a more or less equal number of tasks is selected, because it is not known how high the workload is, relative to the other workers. This is done by taking a random worker that is stored in the list of receiving workers and choosing every task in the list where $i \mathbf{modulo} w_{receiving} = 0$. For the next worker this is $k + (i \mathbf{modulo} w_{receiving}) = 0$. The tasks selected are then assigned to the receiving worker id.

CHAPTER 6

Evaluation

6.1 Setup

In this section the setup for the different experiments is discussed. In subsection 6.1.1 the program that will be used for the experiments is described. Then in subsection the parameters that are used by the placement schedulers are outlined. And finally in subsection 6.1.3 the different experiments that are done and the values used for the parameters are presented.

6.1.1 Raytracer

The raytracer algorithm [14] is an algorithm which is ideal for parallelization. It is used in computer graphics to create a 2 dimensional image out of a 3 dimensional scene. Every pixel beams a ray into the scene and returns the color that the ray crosses first in the scene. The raytracing can be done independently for each pixel and thus it can be done in parallel.

The SNet implementation for the raytracer can be found in appendix A. It consists of three main parts, a splitter, solvers and a merger. The splitter splits up the data into parts, hands them over to the solvers that do raytracing on the given parts and afterwards the merger collects all data and creates an image.

The raytracer has a number of inputs. The first is the scene. Then there are the number of nodes, tasks and tokens. The number of nodes defines the number of solvers that are running concurrently. The number of tasks is used by the splitter to determine how to split up the data in the scene. It creates a number of data packages which consist of an equal amount of work that are passed to the solver. The tokens can be ignored in a Single Memory Processor setup as this is only used in distributed S-Net [6] in this case and is set equal to the number of tasks [14].

6.1.2 Parameters placement schedulers

As already described there are two placement schedulers, which are both used for experiments. Both schedulers have a threshold parameter. For the random scheduling algorithm this threshold defines the chance that a task migrates. The higher the threshold the lower the chance a task migrates.

The ready-time scheduling algorithm uses the threshold to determine if the waiting time is long enough for it to migrate. This is compared to the average waiting time of all tasks for a worker. The threshold value is multiplied by this average and if the task has a longer ready-time it will be put in the waiting queue.

Other parameters used are:

- The number of workers that should run the box and control tasks.
- A parameter for programs running separate workers for box tasks and separate workers for control tasks. The number defines the amount of workers assigned to run control tasks.

- A flag that defines if priorities are used for box and control tasks, where boxes have a lower priority.

6.1.3 Experiments

There are a number of experiments done to compare the different runtime implementations and placement algorithms. This also contains the runtime implementation prior to this research. To do these experiments, a time measure is used that is build into the raytracing program. It uses the function `clock_gettime` to measure the time between start and finish of the program. Timings on this program are done after the run-time system has been initialized and before the run-time system closes down. In the table below are all the parameters that are used for these experiments:

Random scheduling algorithm	
Threshold	0.5, 0.75, 0.9, 1
Ready-time scheduling algorithm	
Threshold	0.9 1 1.05 1.1
Raytracing	
Number of solvers	1, 2, 4, 6 , 8, 10, 11
Number of workers	1, 2, 4, 6, 8, 10, 12
Number of workers control tasks	1, 2, 3, 4, 5, 6, 7, 8
Number of reruns	3
Measure precision in seconds	0.4000250

A combination of these parameters is used also with and without using priority scheduling and task classification.The precision of the measurments is the amount of deviance the measurement can have. All experiments are done on a shared memory machine, a 12 core Intel(R) L5640 2.27 GHz Xeon(R) CPU, with 24 Gigabytes of RAM.

6.2 Results

In figure 6.1 a comparison is shown between four different implementations of the runtime system, the random placement algorithm, the ready-time placement algorithm, the previous S-Net implementation without task migration using LPEL and the Pthread implementation (which does not run LPEL). The old implementations perform better for each worker setting. One note to make is that the performance does not improve when placement scheduling is used using two workers compared to using one worker. The reason for this could be the initial placement strategy and the lack of migration when a small number of workers is used.

Furthermore, in figures 6.2 and 6.3 barplots are shown that compare the performances of the different algorithms. It compares the performance without the use of a priority queue and with the use of a priority queue. The random placement algorithm does not have consistent results. The ready-time placement algorithm shows that from six workers the thresholds 0.9 and 1 are performing better than higher thresholds. This shows that when chances are higher for migration the performance is getting better. Also, it performs better than the random scheduler and thus can be concluded that the information used by the read-time placement algorithm has some benefits on the performance. The difference between using a priority queue and not using one, does not seem to matter. Also note that a threshold of one for the random placement algorithm does not do any migrations and the performance is not better than when the placement scheduler does migrate tasks. This implies the initial placement strategy implemented in this research does not perform as expected. Figures 6.4, 6.5, 6.6 and 6.7 show all the performances again for the random placement algorithm and ready-time placement algorithm, in 3d plots.

In figure 6.8 the performance is plotted using classification for the ready-time placement algorithm with threshold 0.9 and 1 and the random placement algorithm with threshold 0.5 with 12 workers. On the x-axis there are the number of workers used for control tasks, the total number of workers is twelve and the number of workers for box tasks is twelve minus the number of workers for control tasks. The ready-time placement algorithm starts to perform better when more workers are assigned to run only control tasks. Reasons for this are unclear.

Figure 6.9 shows the speedup for the random placement algorithm with threshold 0.5, the ready-time placement algorithm with threshold 0.9 and runtime system prior to this research. The speedup is calculated as follows: $Speedup_p = \frac{Time_1}{Time_p}$ where p is the number of cores on which the program runs, $Time_1$ is the measurement of the S-Net and LPEL implementation without task-migration on one core. In figure 6.10 the efficiency is shown for the the three different runtime implementations. The efficiency is calculated as follows: $Efficiency_p = \frac{Speedup_p}{p}$. The old implementation shows a linear decline in efficiency. From 2 workers on the ready-time placement algorithm has a linear increase in speedup. The random placement scheduler shows that there is also inconsistency in the speedup.

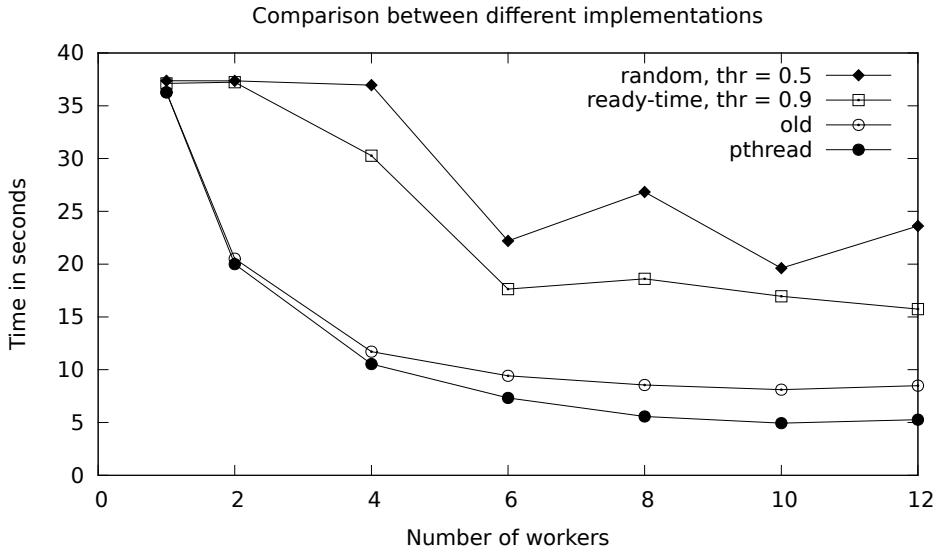


Figure 6.1: This graph shows the performances of the different placement scheduler algorithms compared to the old S-Net and LPEL implementation and the old S-Net pthread implementation. The x-axis defines the number of workers used by the program and the y-axis is the time from the start of the raytracing program until the end. The startup and cleanup of S-Net and LPEL runtime system is not included in this measurement.

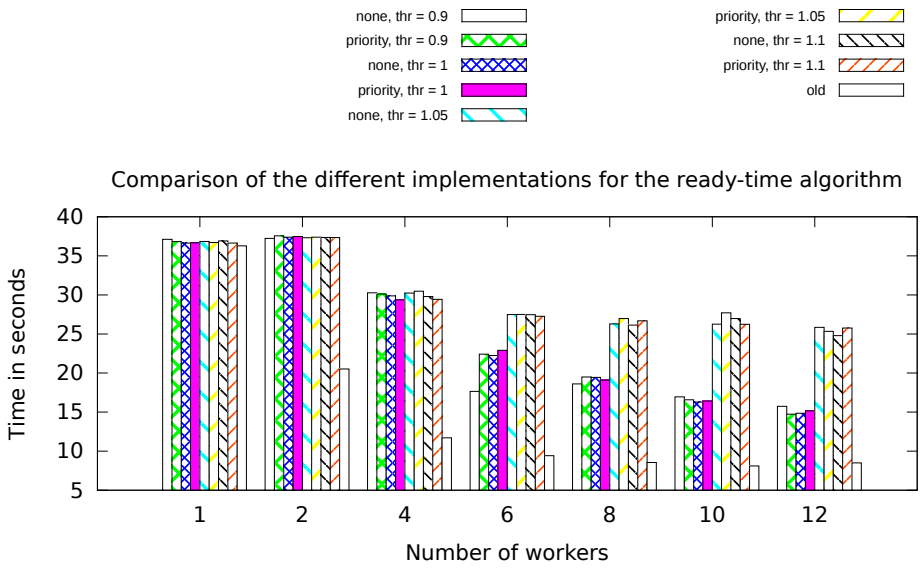


Figure 6.2: This barplot shows the performances for the ready-time placement algorithm. This plot compares the implementation with priority with the implementation without. Old is the previous S-Net and LPEL implementation that did not use task-migration

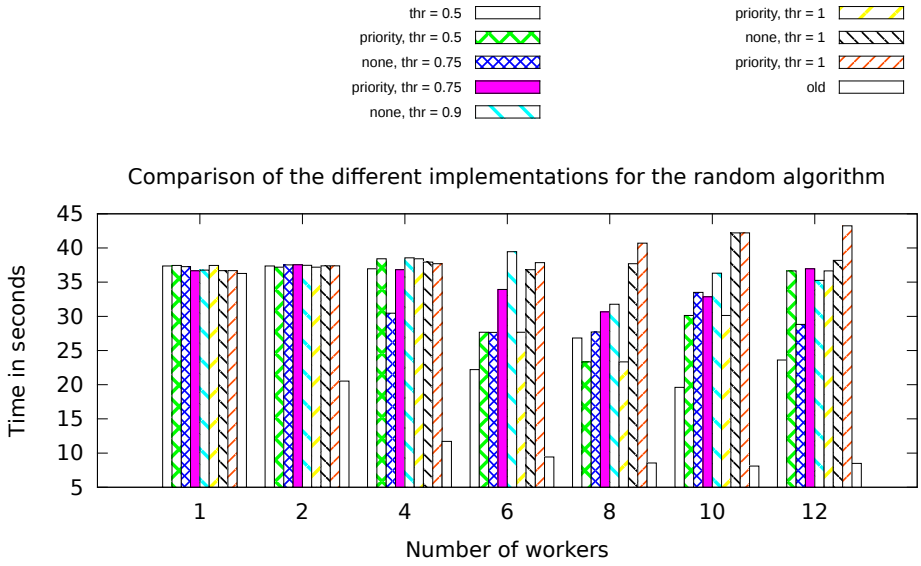


Figure 6.3: This barplot shows the performances for the random placement algorithm. This plot compares the implementation with priority with the implementation without. Old is the previous S-Net and LPEL implementation that did not use task-migration

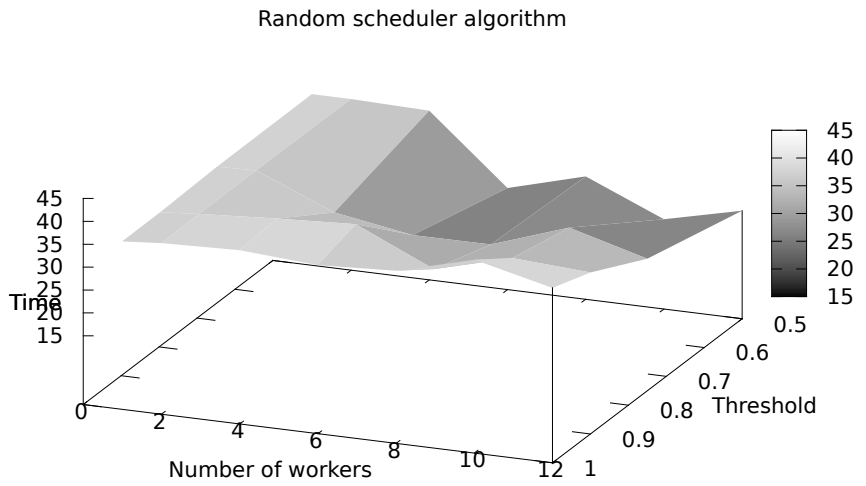


Figure 6.4: This 3d graph shows the performance of the Random placement algorithm. The thresholds used are 0.5, 0.75, 0.9 and 1 (1 implies that tasks will not migrate). The other axis defines the number of workers. The z-axis shows the performance of the algorithm for the raytracing program. The startup and cleanup of the S-Net and LPEL runtime system is not included.

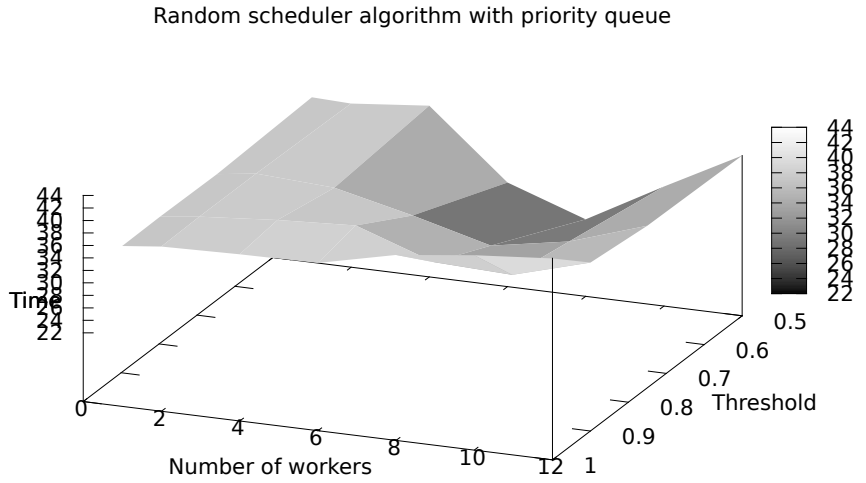


Figure 6.5: This 3d graph shows the performance of the Random placement algorithm using a priority queue with priority 1 for control tasks and priority 0 for box tasks. The thresholds used are 0.5, 0.75, 0.9 and 1 (1 implies that tasks will not migrate). The other axis defines the number of workers. The z-axis shows the performance of the algorithm for the raytracing program. The startup and cleanup of the S-Net and LPEL runtime system is not included.

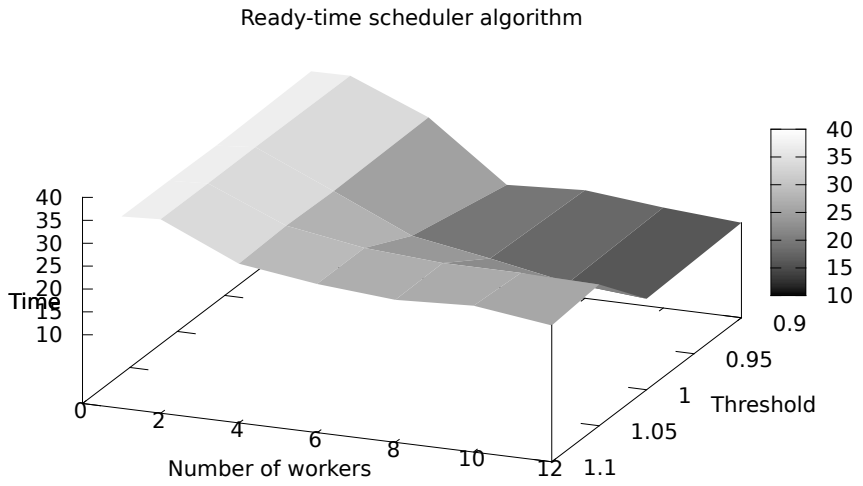


Figure 6.6: This 3d graph shows the performance of the Ready-time placement algorithm. The thresholds used are 0.9, 1, 1.05 and 1.1. The other axis defines the number of workers. The z-axis shows the performance of the algorithm for the raytracing program. The startup and cleanup of the S-Net and LPEL runtime system is not included.

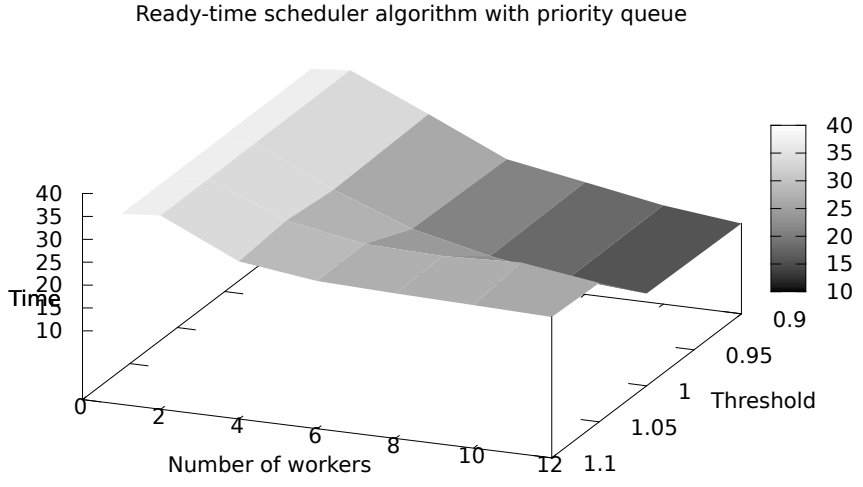


Figure 6.7: This 3d graph shows the performance of the Ready-time placement algorithm using a priority queue with priority 1 for control tasks and priority 0 for box tasks. The thresholds used are 0.9, 1, 1.05 and 1.1. The other axis defines the number of workers. The z-axis shows the performance of the algorithm for the raytracing program. The startup and cleanup of the S-Net and LPEL runtime system is not included.

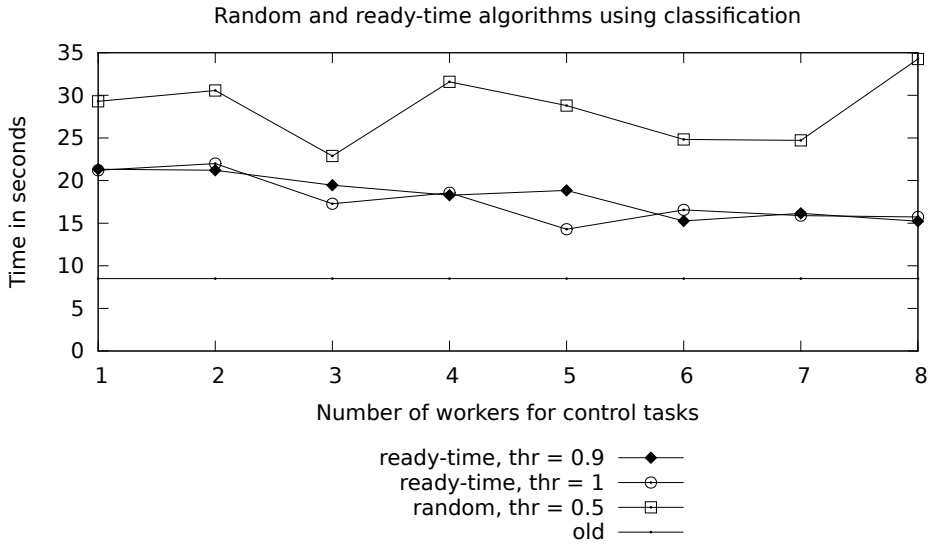


Figure 6.8: Classification using the ready-time and random placement algorithms with 12 workers. The x-axis defines the number of workers running control tasks, the number of workers for box tasks is the total number of workers minus the number of workers for control tasks. The y-axis is the amount of time measured from the start of the raytracing program up to the end. The startup and cleanup of the S-Net and LPEL runtime system is not included.

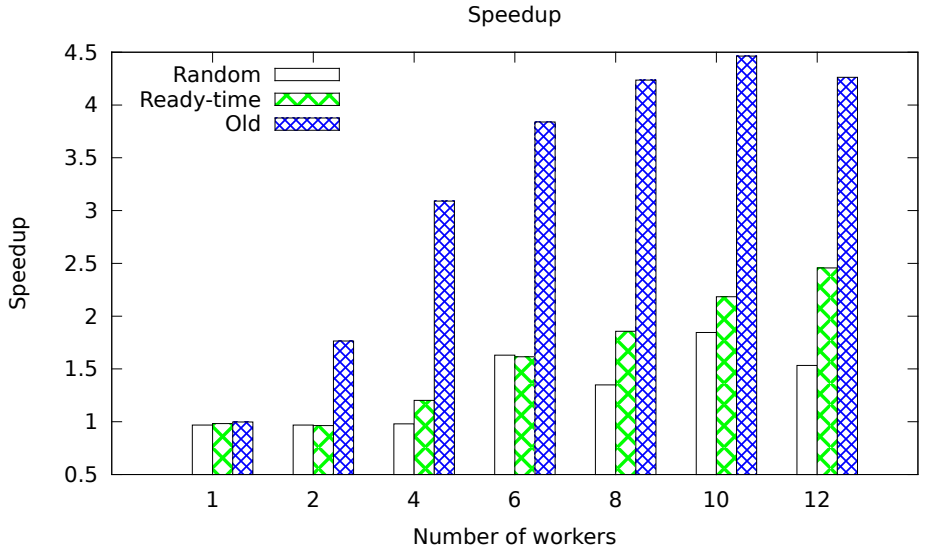


Figure 6.9: This barplot shows the speedups compared to the previous S-Net and LPEL implementation using one worker. The previous S-Net and LPEL implementation is also plotted as a comparison next to the random placement algorithm with threshold 0.5 and ready-time placement algorithm with threshold 0.9. The speedup is calculated as follows: $Speedup_p = \frac{Time_1}{Time_p}$ where p is the number of cores (and workers) that are used.

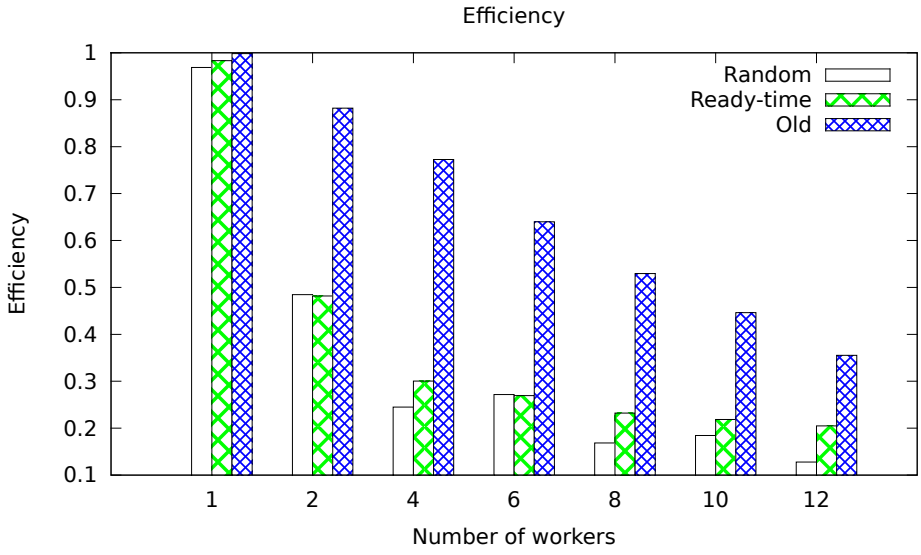


Figure 6.10: This barplot shows the efficiencies compared to the old S-Net and LPEL implementation using one worker. The old S-Net and LPEL implement is also plotted as a comparison next to the random placement algorithm with threshold 0.5 and ready-time placement algorithm with threshold 0.9. The efficiency is calculated as follows: $Efficiency_p = \frac{Speedup_p}{p}$ where p is the number of cores (and workers) that are used.

CHAPTER 7

Conclusion

7.1 Discussion

The results of both placement schedulers are disappointing. They clearly perform worse than the old implementation. The most striking observation to be made is that the performance does not improve when the program runs on two workers instead of one. The reason for this is unclear. Furthermore, the random placement algorithm with a threshold of 1 shows that the initial placement is not working as well as expected. Certainly compared to the the implementation prior to this research, where round-robin is used for initial placement, the results show that locality as used in this research for the inital placement strategy, does not seem to play an important role in the performance on shared memory machines.

This does not mean the use of locality could not improve the performance. The raytracer has a clear preference in its program structure to put solvers on different cores. However, the initial placement does not guarantee every solver will run on a different worker. It could be better to place these solvers on workers that are not already running a solver. Here the placement scheduler could play a role, the solvers could be pinned to some worker or at least have a very low priority to be migrated and other tasks running next to these solvers could be divided by the placement scheduler depending on which workers have less tasks to perform and which have too much tasks to perform. This is already implemented in Distributed S-Net [6], but could also be used for multi-core machines.

Furthermore, looking only at the time a task is ready, could be insufficient information to make a good decision. For example there could be some computation heavy boxes running on a worker and also some control tasks. These control tasks will then have a higher ready time then the boxes. The average ready-time for control tasks is the time that the boxes are running (plus some small control task running time), while for a box the average is taken over the time the other boxes are running and the little time the control tasks are running, one box less is used for the average ready-time which leads to control tasks having a higher ready-time than box tasks and box tasks that are not migrated while control tasks are.

Another comparison to make is how intrusive the implementation of placement scheduler is on the total performance. The placement scheduler had to block the complete scheduler queue of a worker to be able to safely run its behaviour without any race conditions that could arise. Comparing the performances where the program runs on one worker, the differences are not significantly in favour of the previous implementation. This shows that although the placement scheduler does not do any migrations, but does runs its behaviour, blocking the task queue has little effect on the performance.

It is also clear that the raytracer input does not scale with the number of workers. Although this does not explain the bad results for the placement scheduler, it does indicate that the full computing potential of the machine not used.

One succes to note is that although the performance is not as what is expected, the ready-time algorithm performs better than just choosing random tasks the migrate. This means that it is beneficial to use some heuristic in a placement algorithm.

7.2 Future Research

Due to time constraints, it has not been possible to implement all the ideas. One aspect in task migration that could be further researched is the knowledge about the program. Using the content of records for example may further help in making a good decision about where to run a task. Also locality is an aspect that has not been examined yet. Instead of migrating a single box task it might be interesting to look at the performance of migrating parts of the network that are close to the box task. Furthermore, as already disccused in section 7.1 certain parts of a network could be pinned to a worker. In the raytracer example, the solvers could all be placed on different workers to divide the heavy work over the different workers.

Another aspect that has shown importance in the results is the initial placement. Further investigation should be done on what the influences are on initial work distribution. For now round-robin performs better than what we implemented, the question is, is a method like round-robin that arbitrarily distributes the work indeed a better implementation instead of using heuristics about the program.

Moreover, distributed S-Net [6] has not been discussed here, but is also an interesting topic for this research. As with migration over different cores, migration over different nodes to balance the work might also increase the performance. One additional problem for distributed S-Net is the communication time and moving complete parts of the network instead of one task might be an interesting implementation for distributed S-Net.

Bibliography

[1] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen, "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, pp. 325 –340, mar 2001.

[2] I. Bell, N. Hasasneh, and C. Jesshope, "Supporting microthread scheduling and synchronisation in cmps," *International Journal of Parallel Programming*, vol. 34, pp. 343–346, 2006. 10.1007/s10766-006-0017-y.

[3] L. Spracklen and S. Abraham, "Chip multithreading: opportunities and challenges," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 248 – 252, feb. 2005.

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[5] C. Grelck, S. Scholz, and A. Shafarenko, "Asynchronous Stream Processing with S-Net," *International Journal of Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[6] C. Grelck, J. Julku, and F. Penczek, "Distributed s-net: Cluster and grid computing without the hassle," in *Cluster, Cloud and Grid Computing (CCGrid'12), 12th IEEE/ACM International Conference, Ottawa, Canada*, IEEE Computer Society, 2012.

[7] M. Verstraaten, "High-level programming of the single-chip cloud computer with s-net," masters thesis, University of Amsterdam, February 2012.

[8] F. Penczek, J. Julku, H. Cai, P. Hölzenspies, C. Grelck, S.-B. Scholz, and A. Shafarenko, "S-net language report version 2.0," Tech. Rep. 499, University of Hertfordshire, School of Computer Science, April 2010.

[9] C. Grelck and S.-B. Scholz, "Sac: a functional array language for efficient multi-threaded execution," *Int. J. Parallel Program.*, vol. 34, pp. 383–427, Aug. 2006.

[10] D. Prokesch, "A light-weight parallel execution layer for shared-memory stream processing," master's thesis, Technischen Universität, Wien, February 2011.

[11] C. Grelck and F. Penczek, "Implementation Architecture and Multithreaded Runtime System of S-Net," in *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers* (S. Scholz and O. Chitil, eds.), vol. 5836 of *Lecture Notes in Computer Science*, pp. 60–79, Springer-Verlag, 2011.

[12] B. Tika, "Work-distribution and load-balancing in distributed and parallel systems," July 2011. Bachelors thesis.

[13] C. Lu and S. ming Lau, "A performance study on load balancing algorithms with task migration," in *In Proceedings, IEEE TENCON*, pp. 357–364, 1994.

[14] C. Grelck, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, and N. Bagherzadeh, "S-net: Separation of concerns in message driven programming,"

[15] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[16] C. Grelck, "The essence of synchronisation in asynchronous data flow," in *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*, IEEE Computer Society Press, 2011.

[17] "S-net home." `http://www.snet-home.org/?page_id=7`.

[18] J. Robinson, S. Russ, B. Heckel, and B. Flachs, "A task migration implementation of the message-passing interface," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, (Washington, DC, USA), pp. 61–, IEEE Computer Society, 1996.

[19] M. S. Squillante and R. D. Nelson, "Analysis of task migration in shared-memory multiprocessor scheduling," *SIGMETRICS Perform. Eval. Rev.*, vol. 19, pp. 143–155, Apr. 1991.

[20] "Pthreads." `https://computing.llnl.gov/tutorials/pthreads/`.

APPENDIX A

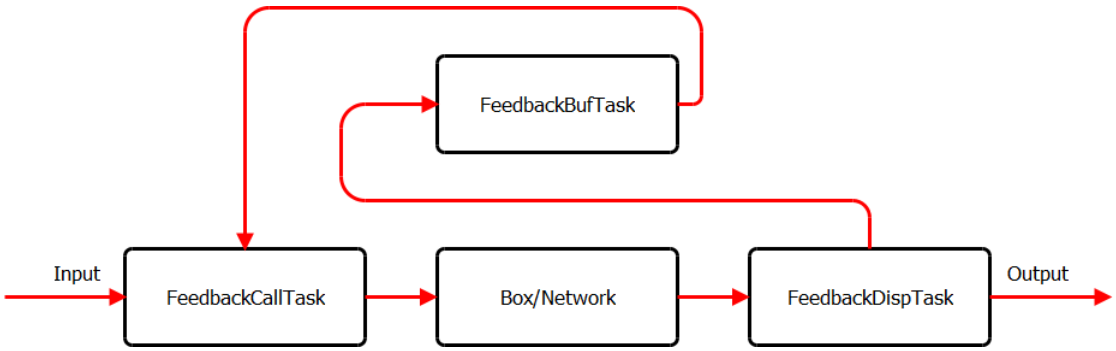
Raytracing

```
net raytracing_smp_dyn( {scn_name, <x_res>,<y_res>,<nodes>,<tasks>,
                         <tokens>,<scheduler>,<done>,<rec>} -> ...)
{

  box toStr( (scn_name) -> (scn_name));
  box t_start(() -> (time));

  box splitter( (<nodes>, <tasks>, <tokens>, <x_res>, <y_res>, <scheduler> )
             -> (<nodes>, <tasks>, <task>, <node>, <x_res>, <y1>, <y2>, <y_res>,
                    <round>, <first>)
              | (<nodes>, <tasks>, <task>, <node>, <x_res>, <y1>, <y2>, <y_res>,
                 <round> )
              | (<nodes>, <tasks>, <task>, <x_res>, <y1>, <y2>, <y_res>,
                  <round> ));

  box solve( (<x_res>, <y1>, <y2>, <y_res>, scn_name)
              -> (<x_res>, <y1>, <y2>, <y_res>, r, g, b, scn_name));

  net merge( {<tasks>, <x_res>, <y1>, <y2>, <y_res>, r, g, b }
             -> {<tasks>, <count>, <x_res>, <y_res>, R, G, B },
             {<tasks>, <x_res>, <y1>, <y2>, <y_res>, r, g, b ,<first>}
             -> {<tasks>, <count>, <x_res>, <y_res>, R, G, B }
           )
  {
    box init((<x_res>, <y1>, <y2>, <y_res>, r, g, b ,<first>)
             -> (<x_res>, <y_res>, R, G, B));
    box merger(( <x_res>, <y1>, <y2>, <y_res>, R, G, B, r, g, b)
                -> (<x_res>, <y_res>, R, G, B));
  } connect
      ( ( init .. [ {<tasks>} -> if <tasks == 1>
                                 then {<tasks>, <count=1>, <done>}
                                 else {<tasks>, <count=1>}        ] )
        | [] )
      .. ( [|{ R, G, B},{ <x_res>, <y1>, <y2>, <y_res>, r, g, b }|]
           ..(
               (merger
                .. [ {<count>} -> {<count = count+1>}]
                .. [ {<tasks>, <count>} -> if <tasks == count>
                                           then { <tasks>, <count>, <done>}
                                           else { <tasks>, <count>}        ])
```

```
              | [])
          )*{<done>} ;

  box genImg((R, G, B, <x_res>, <y_res>, scn_name, time) -> ());
} connect
    toStr
    .. t_start
    .. splitter
    .. [ {<done>} -> {}]
    .. ( ( ( solve
             .. [ { <tasks>, <task>, <node>, <x_res>, <y1>, <y2>, <y_res>,
                    r, g, b, scn_name }
                  -> { <tasks>, <task>, <x_res>, <y1>, <y2>, <y_res>,
                       r, g, b, scn_name, <done> };
                     { <node> } ]
           )!@<node>
           | []
         )
         .. ( [ {<done>, <task>} -> {<done>,<task>} ]
              | [| {<task>}, { <node>} |]
            )
       ) * {<done>}
    .. [ {<done>} -> {} ]
    .. merge
    .. genImg;
```

APPENDIX B

Implementation feedback combinator



The feedback combinator is implemented in a different way as the other combinators as it consists of multiple tasks. This chapter gives a small explanation what the feedback combinator looks like.

The feedback combinator is divided into three different tasks. The first is `FeedbackCallTask` which receives records from the input stream and records coming back from the feedback combinator. It sends the record to the box task, or to the lower network and the output of this box or network is send to `FeedbackDispTask`. This task decides whether the output record is send to the output stream or back into the feedback combinator. If it goes back to the input of the feedback combinator, it is first send to the third task `FeedbackBufTask` which sends it back to the `FeedbackCallTask`.

APPENDIX C

Implementation changes

- `Init{FunctionName}`: This is a new function for each entity (box and control tasks). The function initializes some argument values, like the stream descriptors that are used by this task.
- `lpel_task_t`: This structure has been extended with a few extra variables. The most important variables are the integers `current_worker` and `new_worker` which containt the indices of the current worker the task is running on and the worker the task should migrate to. If they are the same, the task just goes on running.
- `EntityTask` This function has been changed. It is called by LPEL to run an entity. Now the implementation is extended with a loop. The entity structure given as a parameter contains a boolean. Before running this is set to zero, then the task is called. And if the boolean is not set to true, the function stops and does a cleanup. The precise implementation can be found in appendix **??**.
- `SNetThreadingInitSpawn` This function is added to provide the initial spawn of a task. Tasks that migrate, are newly created LPEL tasks, but continue computations done by a previous task.
- `SNetThreadingReSpawn` This function is called after an entity task has finished an iteration and has not terminated. It either sets the boolean used in `EntityTask` to true if the task should keep running on the same worker, or create a new LPEL task on a different worker.
- `SNetThreadingInitialWorker` This function decides on which worker the entity should be created. The second argument is an integer which defines what type the type of situation is for that instance. Boxes are never set on a different worker than its parent so they initialize with type 0. Most control tasks initialize with 1 which keeps the control task on the same worker as its parent. For 2 the entity is put on a next worker. And finally 3 will put the entity on the current worker.
- `LpelStreamRead`, `LpelStreamWrite`, `LpelStreamPoll` These functions have been changed to make task migration more efficient. Stream descriptors are bound to a task, this is due to the fact that when there is a read or write on a stream and the task has to wait on the I/O, the task has to be switched and put on hold. The stream descriptor is used to determine which task to put on hold. In the new implementation, a task can be spawned and a new `lpel_task` is created, making the stream descriptor useless. To cope with this, the function `LpelTaskSelf` is called in the given functions to determine which task is used in the stream descriptor.
- `LpelTaskCreate` This function has an extra parameter to set the priority of the task.